1988 International Conference on Parallel Processing, August 1988.

LOOKAHEAD IN PARALLEL DISCRETE EVENT SIMULATION

Richard M. Fujimoto[1]
Computer Science Department
University of Utah
Salt Lake City, UT 84112

Abstract

Empirical performance evaluations of parallel, discrete event simulation algorithms using deadlock avoidance and deadlock detection and recovery techniques developed by Chandy and Misra have been performed using the BBN Butterfly™ multiprocessor. Experiments using synthetic workloads reveal that the degree to which processes can look ahead in simulated time plays a critical role in the performance of distributed simulators using these algorithms. These results are applied to a queueing network simulation where as much as an order of magnitude improvement in performance is observed if the distributed simulator is programmed to fully exploit the lookahead available in the application. Performance measurements of several hypercube-based communication network simulators provide additional empirical data to support these claims. These results demonstrate that substantial improvements in performance are obtainable if the application can be programmed to have good lookahead characteristics. On the other hand, other applications *inherently* contain poor lookahead properties, and appear to be ill-suited for these simulation algorithms.

Keywords: [illegible] network simulation. (KR) ←

1. Introduction

Discrete event simulation has long been a task with computation requirements that challenge the fastest available computers. For example, simulations of communication networks, parallel computer architectures, and battlefield scenarios often require hours, days, or even weeks of CPU time using traditional, single processor techniques. Simulator performance may be improved using vectorizing techniques [Chan83a], processors dedicated to specific simulation functions [Comf84a], execution of independent trials on separate processors [Bile85a], or the execution of a single instance of a simulation program on a parallel computer. The last technique, referred to as distributed simulation, is the subject of this paper.

Simulation would initially appear to be a natural candidate for parallel processing because many of the aforementioned applications contain a high degree of parallelism. However, the exploitation of this parallelism is elusive because the global notion of simulated time does not easily map onto a distributed computer. This property distinguishes distributed simulation from other forms of parallel computation.

Several schemes have been proposed to solve this problem. A survey of the literature has been reported by Kaudel [Kaud87a]. One important class of distributed simulation algorithms is the so-called "conservative" mechanisms. Chandy and Misra developed a mechanism based on a deadlock avoidance technique where null messages are used to distribute clock information among the processes taking part in the simulation [Chan79a, Misr86a]. Another mechanism, also developed by Chandy and Misra, is based on a deadlock detection and recovery paradigm — the simulator runs until deadlock, the deadlock is detected, and an algorithm is executed to break the deadlock [Chan81a, Misr86a]. Other approaches to distributed simulation have been proposed, notably the Time Warp approach proposed by Jefferson [Jeff85a], but the work discussed here will be confined to deadlock avoidance and deadlock detection and recovery techniques.

In [Fuji88a] several experiments using synthetic workloads were described that were designed to evaluate the effectiveness of distributed simulation strategies using the deadlock avoidance and the deadlock detection and recovery algorithms. These experiments were performed on a distributed simulation testbed that was implemented on the BBN Butterfly,™ a shared-memory multiprocessor. Here, we apply these results to specific application problems to provide empirical data to support these results. In particular, parallel simulations of queueing networks and the communication subsystem of a hypercube-based multicomputer demonstrate the relationship between lookahead in the simulation application and performance of the parallel simulator.

[1]This work was supported by ONR contract number N00014-87-K-0184 and NSF grant number DCR-8504826

2. Logical Processes, Activities, and Lookahead

Logical processes, activities, and lookahead form the basis for the synthetic workload model that is used here. The simulation program consists of some number of *logical processes*, each of which models some portion of the system being simulated. For example, in simulating a digital logic network, each gate (or some collection of gates) could be modeled by a logical process. Logical processes communicate exclusively by exchanging timestamped messages. Messages typically correspond to events that trigger a change in system state. Each logical process must process incoming messages in non-decreasing timestamp order to ensure that cause-and-effect relationships are faithfully reproduced by the simulator.

We informally define an *activity* as a sequence or thread of events that propagates among the logical processes in the simulation. These events model some sequence of cause-and-effect relationships in the system being simulated. For example, in a logic simulation, individual events are logic signal transitions and each activity corresponds to a signal propagating through a sequence of logic gates. In a queueing network simulation, each activity corresponds to a job traveling through the network. Activities are usually dynamic. A new activity is created in the logic simulation whenever an existing activity reaches a fanout point in the network. The activity disappears when (for instance) it reaches an AND gate with a logic zero on one of the other input lines. For our purposes, this informal definition of activities and logical processes will suffice.

Logical processes often "look ahead" into the simulated time future to schedule new events. For example, upon receiving a signal transition event in a logical process for an inverter gate, the process can predict and schedule a new event (a signal transition at the output of the gate) one gate delay later in simulated time. The lookahead abilities of the process determine how readily it will schedule new events. Processes such as the inverter with *good* lookahead abilities can "see" sufficiently far into the future that "effect" events can be scheduled as soon as the "cause" event is received. On the other hand, processes with poor lookahead ability must first wait until simulated time is advanced before they can schedule the effect event. For example, in a queueing network simulation with prioritized jobs, the "departure" event for a low priority job cannot be scheduled until it is first determined that no higher priority job will preempt it.

Quantitatively, lookahead is defined as follows: if a process has knowledge of all events that will occur up to simulated time T, and can predict all new events it will generate with timestamp $T+L$ or less, then the process is said to have lookahead L. In general, lookahead is a complex function that varies with time and the type of event, and is highly dependent on details of the simulation problem and the way it is programmed. A process can schedule a future event so long as the timestamp on that event is less than or equal to the process's local clock plus its lookahead. Such events are said to be within the "lookahead horizon" of the process.

Consider a "cause" event with timestamp T_{cause} that leads to an "effect" event with timestamp T_{effect}. The *absolute* value of lookahead is not as important as the lookahead *relative* to $T_{effect} - T_{cause}$, because this will determine how far the process must advance in simulated time to generate the new event. Therefore, we define a quantity referred to as the *lookahead ratio (LAR)*:

$$LAR = \frac{T_{effect} - T_{cause}}{lookahead}.$$

A *low* (e.g., 1.0) *LAR* corresponds to a *high* degree of lookahead.

3. The Distributed Simulation Testbed

An 18 processor BBN Butterfly multiprocessor was used for experimentation. Each processor node contains a 16 MHz MC68020 with MC68881 floating point coprocessor, 1 to 4 MBytes of memory, and a

Table 1. Hardware Parameters

Operation	Execution Time (microseconds)
Local memory reference	0.60
Remote memory reference	4.0
Register-to-register instruction	0.71
16 bit Load (Local Memory)	1.3
16 bit Load (Remote Memory)	6.3
Parameterless function call	6.9
Atomic inclusive OR	20

processor node controller (PNC), a microcoded engine that processes local and remote memory requests. The interconnection switch is configured as an Omega network. Atomic test-and-set like memory operations are also implemented in the PNC. Execution times of various instructions and operations are shown in table 1. Experimental data indicate that switch contention, and hot spot congestion in particular, is unlikely [Thom86a].

Each processor executes a single operating system process. This process is a scheduler that time multiplexes execution of the simulation processes mapped to the processor. This strategy avoids excessive context switching overhead, and allows more direct control over the process scheduling mechanism. Asynchronous message passing primitives were constructed using direct memory accesses to the mailbox in the receiving simulator process. Only a few simple Butterfly primitives, namely lock and atomic-add operations, are used by the testbed after initialization is complete.

4. The Simulation Algorithms

Two distributed simulation algorithms were implemented in the testbed: one based on deadlock avoidance and another based on deadlock detection and recovery. The shared memory architecture of the Butterfly was used to improve the efficiency of these algorithms, as described below. A single processor, event list implementation was also developed in order to compute speedup.

4.1 Deadlock Avoidance Strategy

The deadlock avoidance scheme developed by Chandy and Misra was implemented first. Each logical process sends a null message to each of its neighbors whenever it blocks. The timestamp on this message represents a lower bound of the timestamp on any message that will be sent to the receiver in the future. It is equal to the local clock value of the process plus the lookahead value because, by definition, the process cannot predict the occurrence (or non-occurrence) of events further into the future. Chandy and Misra have shown that this approach is sufficient to avoid deadlock [Chan79a].

In the testbed, one optimization was performed to streamline the processing of null messages. Rather than enqueueing each null message sent to another processor, a single variable is associated with each input link that contains the timestamp of the last null message that was received. This avoids unnecessary enqueue and dequeue operations and leads to more efficient memory utilization.

4.2 Deadlock Detection and Recovery Strategy

The second simulation approach is based on deadlock detection and recovery. The simulation runs until deadlock, the deadlock is detected, and an algorithm is initiated to break the deadlock [Chan81a]. A central controller is used to coordinate the deadlock recovery procedure.

Deadlock in the testbed is easily detected by maintaining a global counter indicating the number of processes that are either scheduled or running. The system is deadlocked whenever the counter reaches zero and there is at least one process that has not yet terminated (otherwise, the computation has terminated). Each scheduler checks the deadlock counter whenever it fails to find a process to run, and initiates a computation to break the deadlock if it finds the counter is zero.

The deadlock recovery algorithm locates the message in the system with the smallest timestamp and arranges for it to be processed next. A distributed algorithm is used to perform this computation. A central controller is used to coordinate this activity. By convention, the scheduler executing on PE 0 acts as the controller.

An alternative deadlock recovery algorithm was also implemented in which messages are propagated throughout the system in order to restart as many processes as possible. This algorithm is described in [Chan81a]. It was found, however, that the additional time required to execute this algorithm yielded a net loss in performance. The performance figures reported here are based on the former deadlock recovery approach.

4.3 Uniprocessor Simulation Algorithm

Finally, a single processor, event list simulator was developed to allow comparison of distributed simulation programs with sequential event list implementations. In order to obtain a fair comparison, the uniprocessor simulator was constructed by modifying the distributed simulator. Both implementations maintain the same overall structure, organization, programming style, and conventions. All code specific to parallel computation (e.g., synchronization locks) was eliminated.

The event list was implemented as a splay tree [Slea85a]. Empirical evidence suggests that splay trees are among the fastest methods for implementing an event list [Jone86a]. An alternative implementation using a singly linked linear list was also developed. It was found that this implementation yielded performance comparable to the splay tree for small simulations but, as expected, ran much more slowly for the larger simulations. The splay tree implementation is used in all comparisons with uniprocessor simulations reported here.

4.4 Performance Metrics

Three metrics are defined to evaluate the performance of the distributed simulation programs:

- **Speedup.** $SU(n)$, the speedup using n processors, is defined as the execution time of the single processor, event list implementation using a splay tree divided by the execution time of the distributed simulation program when n processors are used.
- **Null Message Ratio.** *NMR* is defined as the number of null messages processed by the simulator using deadlock avoidance divided by the number of real (non-null) messages processed. This measures the overhead of the deadlock avoidance approach.
- **Deadlock Ratio.** *DR* is the number of messages processed by the distributed simulator using deadlock detection and recovery, divided by the number of deadlocks that occur. This figure measures the efficiency of the deadlock detection and recovery algorithm.

The single processor execution times were obtained by running the splay tree simulator on a single node of the Butterfly. The same compiler as that used by the distributed simulator was used. Therefore, compiler and processor speed dependencies are factored out of the speedup figures.

The experiments were performed with no other applications running on the Butterfly. Facilities, such as the window manager, were run on processors different from those executing the simulation program. These measures were taken to minimize interference with the computation.

Experimental data were, for the most part, well behaved. The 95 percent confidence intervals for the measured data were typically less than one or two percent of the reported value. Only in a few instances were significant variations observed from one measurement to another. These were related to the avalanche effect described later, and do not affect the conclusions that follow from these experiments.

5. Experiments Using Synthetic Workloads

Synthetic workloads were constructed based on the notions of logical processes, activities, and lookahead, described earlier. Workloads contained 16 and 64 logical processes organized in 4 by 4 and 8 by 8 toroids, respectively (a toroid is a nearest neighbor mesh with wrap-around edge connections). Toroids were used because they do not contain inherent bottlenecks that might color the results, and because they are rich in cycles, and therefore represent a reasonably challenging configuration for the simulation algorithms. It is assumed that the number of activities in the simulation remains constant, and the lookahead of each process remains fixed throughout the simulation and does not depend on the type of event. Within each experiment, a fixed number of messages (the message population) circulates in a manner similar to jobs traveling throughout a closed queueing network. Simulation activity in each process was emulated using busy wait loops.

The experiments discussed next assume a message population of four messages per process and an average computation time of 1 millisecond (selected from a random variable with a negative exponential distribution) to process each incoming message. A static process to processor mapping

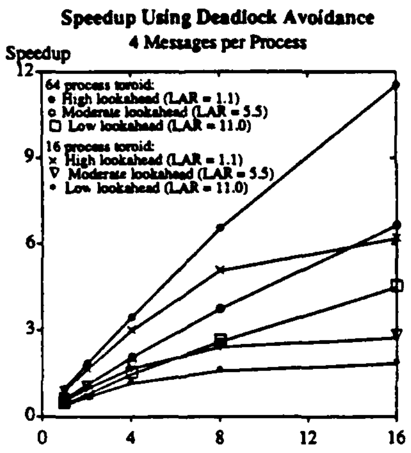

Figure 1. Speedup of synthetic workload as lookahead is varied.

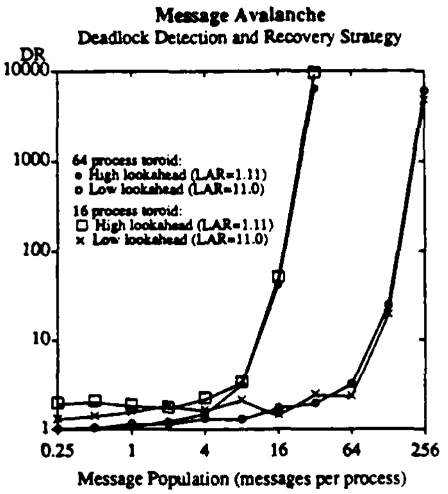

Figure 2. Message avalanche occurs as the message population is increased

was used that balanced the workload assigned to the available processors while minimizing interprocessor communications.

Numerous experiments were conducted to examine the effects of computation granularity, dynamic load balancing, message population, message routing, and other factors. A detailed description of these results is beyond the scope of the present discussion, but is described elsewhere [Fuji87a, Fuji88a]. We will summarize some of these results and discuss how they can be applied to a specific application.

5.1 Effect of Lookahead

The speedup curves in figure 1 show the effect of varying lookahead in the deadlock avoidance simulator. As can be seen, lookahead plays a critical role in determining simulator performance. Performance degrades significantly as the lookahead ability of each process is reduced. Processes with poor lookahead characteristics must delay generating new events, reducing the amount of parallelism available in the simulation.

Performance of the 16 node toroid is somewhat less than the 64 node toroid because the simulation does not contain sufficient parallelism to keep all of the processors busy. In addition, as the number of processes per processor is decreased, each process is afforded less time to collect messages before it is executed by the scheduler. As a result, a process may be scheduled more often than if there were more processes mapped to the processor. The additional scheduling overhead and increased idle time lead to poorer performance in the 16 node simulator, particularly as the number of processors is increased.

5.2 Message Avalanche

Experiments using the deadlock detection and recovery strategy also revealed an "avalanche" phenomenon. This behavior is depicted in figure 2 where the deadlock ratio is plotted as a function of the message population. Performance remains poor (only a few messages processed between deadlocks) at low and moderate message populations, but then increases dramatically once message population reaches a certain critical level. It was found that message avalanche was a prerequisite for achieving good performance for this simulation strategy.

Message avalanche occurs when a message arriving at a process causes the transmission of one or more additional messages, which in turn trigger the transmission of still others, and so on. A multiplicative effect occurs whereby an "avalanche" of message traffic results from the original, accounting for the dramatic improvement in simulator efficiency.

As shown in figure 2, the message population required to induce avalanche was found to be dependent on the lookahead ability of the processes. Smaller populations were required to induce avalanche if processes were able to see far into the simulated future. This is again because poor lookahead characteristics reduce the amount of parallelism in the simulator.

5.3 Processes with Different Lookaheads

The experiments described above used homogeneous workloads where each process behaved in the same way as the others. Many real simulations contain a variety of logical processes with different lookahead characteristics. Additional experiments were performed in which some processes had poorer lookahead characteristics than the others.

Figures 3 and 4 show simulator overhead for the deadlock detection and recovery, and deadlock avoidance simulators, respectively, when some number of processes with poor lookahead characteristics are mixed with processes with good lookahead characteristics. Experiments were performed in which one, one fourth, one half, and finally all processes have poor lookahead (high *LAR*). Figure 3 indicates that the presence of a few processes with poor lookahead results in a perceivable performance degradation in the deadlock detection and recovery simulator (the avalanche point is moved to higher message populations). When a significant fraction of the processes have poor lookahead, performance is almost the same as that when all processes have poor lookahead. The deadlock avoidance simulator was found *not* to be as susceptible to such behavior (see figure 4), though some degradation results if a sufficiently high fraction have poor lookahead properties.

6. Queueing Network Simulations

To illustrate the applicability of the above results in a specific application, queueing network simulations were performed. A five process, central server network was simulated on the testbed. As shown in figure 5, this network contains three first-come-first-serve (FCFS) processes that service incoming jobs in the order in which they arrive, a fork process that stochastically routes each incoming job to one of its output ports (assume for now that either port is equally likely to be selected), and a merge process that combines streams of incoming jobs into a single output stream. Each server process also computes the average number of jobs in the server and reports this figure to the user.

Simulation and empirical studies by Seethalakshmi and Reed respectively concluded that the central server network is ill-suited for the conservative distributed simulation algorithms discussed here [Seet79a, Reed88a]. We reproduce and explain the poor results that these researchers observed in terms of message population and lookahead, and utilize this knowledge to improve performance.

The "classical" implementation of the FCFS process uses two types of events: arrival events (scheduled by other processes) denote jobs arriving at the server, and departure events (scheduled by the FCFS process itself) denote jobs completing service. The actions executed by the server process for each event type are shown in figure 6. *NJobs* indicates the number of jobs currently residing in the server, and *ServiceTime* indicates the time required to service each job. Code for computing statistics is not shown.

The classical server process has very poor lookahead properties. This is because it will not transmit an arrival event message with timestamp *TS*

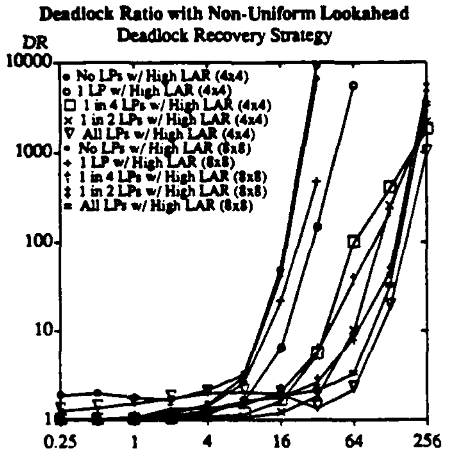

Figure 3. Overhead with non-uniform lookahead — deadlock recovery.

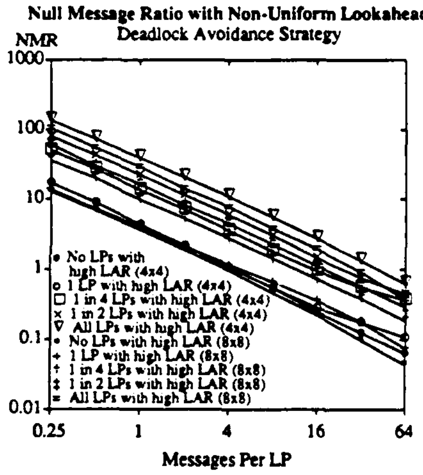

Figure 4. Overhead with non-uniform lookahead — deadlock avoidance.

until it has first advanced its local simulated time clock to *TS* by processing a departure event. In effect, it has a lookahead value of zero.

The lookahead properties of the FCFS process can be improved by eliminating the departure event, and generating a new arrival event as soon as one is received. Because an FCFS queueing discipline is used, the departure time can be determined as soon as the message is received. The optimized program is shown in figure 7. *EndService* denotes the time at which the server process will become idle if no additional jobs are received in the future. This program exhibits very good lookahead abilities because it can schedule events far into the simulated time future.

6.1 Performance Using Identical Servers

Simulators using each of these server programs were developed and executed on the Butterfly testbed. In all of the experiments described below, each logical process was mapped to a separate processor, and static scheduling was used. Service times for server processes were selected either deterministically or from a random variable with a negative exponential distribution.

The resulting speedup and simulator efficiencies for the central server queueing model using the deadlock detection and recovery strategy are shown in figures 8 and 9, respectively. The deadlock avoidance simulator yielded similar speedups. As can be seen, reprogramming the server to have better lookahead characteristics dramatically improves performance. Speedup is improved by as much as an order of magnitude. These results are consistent with those obtained using synthetic workloads.

The performance results of the *classical* server process are qualitatively similar to those reported by Reed and Seethalakshmi. The servers used in

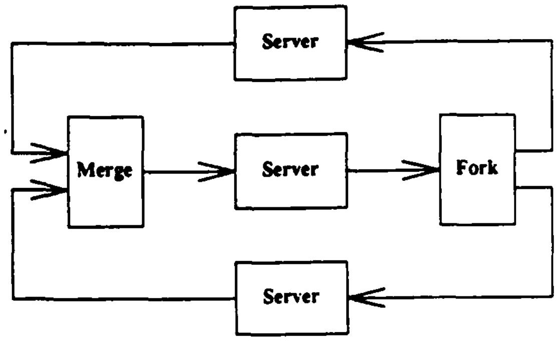

Figure 5. Central server queueing model.

```
ARRIVAL EVENT at TIME T:
    NJobs := NJobs + 1;
    IF (NJobs = 1) THEN  /* if server was previously idle */
        Schedule (local) Departure Event at time T + ServiceTime;

DEPARTURE EVENT at TIME T:
    Schedule (remote) Arrival Event at time T;
    NJobs := NJobs - 1;
    IF (NJobs > 0) THEN  /* if job(s) waiting in queue */
        Schedule (local) Departure Event at time T + ServiceTime;
```

Figure 6. "Classical" program for FCFS server (poor lookahead).

```
ARRIVAL EVENT at TIME T:
    IF (T < EndService) THEN /* if server busy */
        BEGIN
        Schedule (remote) Arrival Event at time EndService+ServiceTime;
        EndService := EndService + ServiceTime;
        END
    ELSE  /* server idle */
        BEGIN
        Schedule (remote) Arrival Event at time T + ServiceTime;
        EndService := T + ServiceTime;
        END
```

Figure 7. Optimized program for FCFS server (good lookahead).

those studies are a variation of the classical server described above, and share the same (poor) lookahead properties — a message will not be forwarded until another message is first received with a timestamp at least as large as the departure time of the first. Therefore, lookahead provides an explanation for the poor performance that they observed.

Although the above results are encouraging, it is important to keep in mind that reprogramming the application to exhibit greater lookahead ability is not always possible. The above optimization relied on the servers using an FCFS scheduling discipline. As we shall soon see, many applications inherently contain poor lookahead properties.

Finally we note that, at first glance, reprogramming logical processes to maximize lookahead may complicate other aspects of the simulation, e.g., statistics collection. For example, the optimized server does not pause for departure events, so statistics that are most easily collected at job departure must be collected at other points in simulated time. This problem is easily reconciled by scheduling local departure events (as was done before) that are only used for statistics collection purposes.

6.2 Performance Using Mixed Servers

Additional experiments were performed to examine the effect of mixing processes with poor and good lookahead characteristics. Recall that experiments using synthetic workloads revealed that a small number of processes with poor lookahead could significantly degrade performance of the deadlock detection and recovery simulator. The deadlock avoidance simulator was found not to be as susceptible to such behavior.

The central server queueing network simulations were repeated where one of the three servers was implemented using the classical server

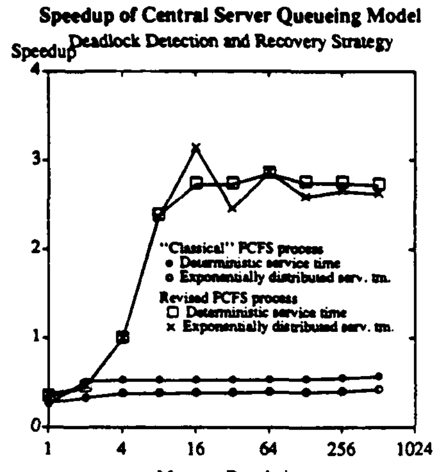

Figure 8. Speedup of central server queueing model.

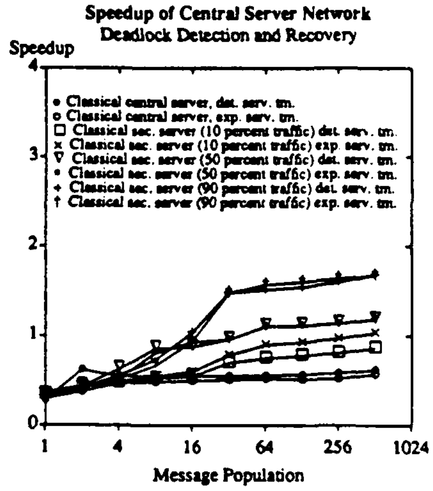

Figure 10. Speedup of detection and recovery simulator with one classical server.

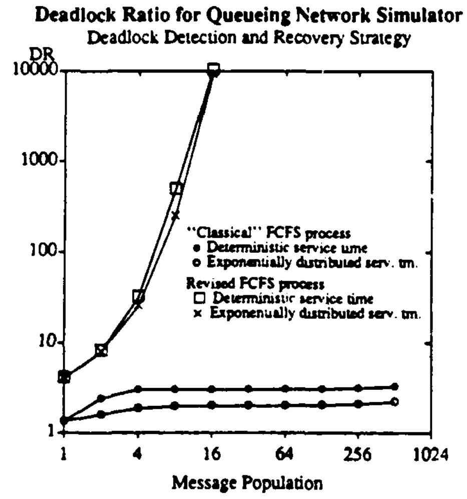

Figure 9. Overhead of central server queueing network simulator.

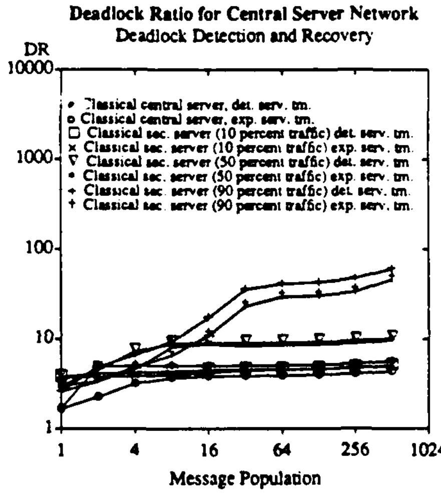

Figure 11. Overhead of detection and recovery simulator with one classical server

program described earlier, and the remaining servers used the optimized program. The resulting simulator is not unlike one that would result if one of the servers was (say) a prioritized queue while the others were FCFS.

The speedup and efficiency of the deadlock detection and recovery simulator is shown in figures 10 and 11. When the central server (the process receiving messages from the merge process) has poor lookahead properties, performance is almost as poor as when *all* of the servers have poor lookahead. When one of the secondary servers (the servers receiving messages from the fork process) has poor lookahead, performance is better, but still well below that of the simulator using only optimized servers. These results are consistent with those obtained using synthetic workloads, and demonstrate that a few processes with poor lookahead can significantly degrade overall performance in the deadlock detection and recovery simulator.

When the classical program was used to implement a secondary server, the routing probabilities in the fork were modified so that 10, 50, and finally 90 percent of the message traffic was routed to the classical server. It is interesting to note that performance improves as *more* traffic is routed toward the server with poor lookahead. If little traffic is directed toward this server, the simulator is constantly deadlocking because the merge process is forced to block because it cannot determine whether or not it is safe to proceed without first receiving a message from this server. Routing additional message traffic toward this server helps the simulator to overcome (somewhat) the server's poor lookahead characteristics.

Speedup and overhead curves for the deadlock avoidance simulator are shown in figures 12 and 13. The deadlock avoidance simulator tends to be more forgiving of processes with poor lookahead. Poor performance results when the central server process has poor lookahead. However, performance begins to approach that of the optimized simulator in some situations where one of the secondary servers has poor lookahead. In particular, good performance is obtained if a significant fraction of the message traffic (50 to 90 percent) is routed *around* the process with poor lookahead. Unlike the deadlock detection and recovery simulator, null message traffic is generated by the classical server to allow the merge process to proceed. Because processes with poor lookahead tend to buffer messages rather than immediately forwarding them, it is best to minimize the amount of traffic routed to the classical server because this only detracts from the available parallelism.

7. Communication Network Simulations

Simulations of the message passing subsystem of a hypothetical multicomputer were also performed. The multicomputer is organized in a hypercube topology, and Sullivan's algorithm is used to route messages to their respective destinations [Sull77a]. Like the queueing network and synthetic workload experiments, a fixed message population was used to control the amount of available parallelism. Initially, each message is assigned a destination to which it is to be routed, and a message length. The destination is selected from a uniform distribution (excluding the

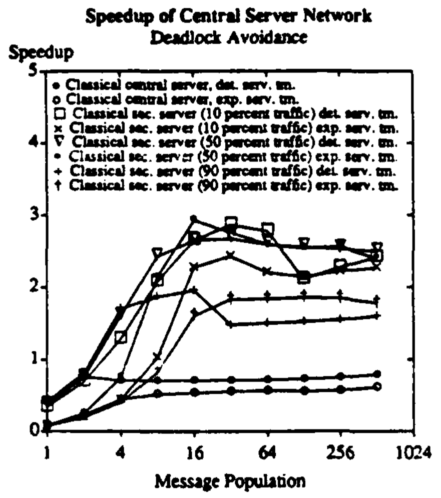

Figure 12. Speedup of deadlock avoidance simulator with one classical server.

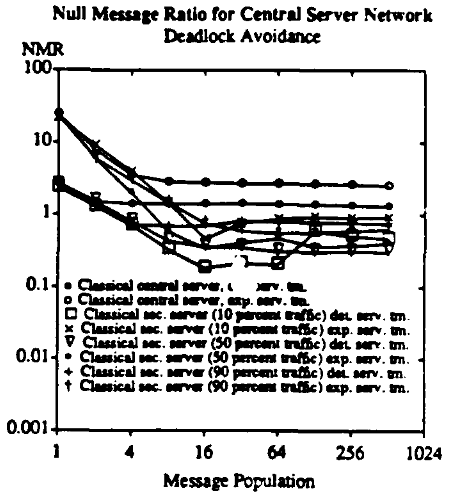

Figure 13. Overhead of deadlock avoidance simulator with one classical server

processor where the message initially resides), and the message length is selected from an exponential distribution. When a message reaches its final destination, a new destination and message length are selected. All communication links in the hypercube are assumed to provide the same bandwidth. Three simulators were developed that contain varying degrees of lookahead, as will be described next.

7.1 A Simulator with High Lookahead

FCFS is a simulator in which messages are simply forwarded on the output link selected by the routing algorithm in FCFS order. Like the FCFS queueing network described earlier, this simulator has great lookahead ability because messages arriving at a logical process (with timestamp denoting the arrival time in the hypercube) can be immediately forwarded.

7.2 A Simulator with Moderate Lookahead

PRIO is a simulator with intermediate lookahead properties. Here, messages are classified as either high priority or low priority. Communication links in the hypercube give preference to high priority messages when selecting the next message to be transmitted. A low priority message is only forwarded if there are no high priority messages waiting to use the link. Messages within each priority level are processed in FCFS order. Each message is assigned a new priority whenever a new destination address and message length are selected and maintains this priority until it reaches the destination processor.

No preemption occurs in this simulator. Once the link begins forwarding a low priority message, it will continue to send it, even if a high priority message arrives before transmission is complete.

The parallel simulator for this system has intermediate lookahead properties. Logical processes have excellent lookahead for high priority messages, but poorer lookahead for those with low priority. Just as is the case for the *FCFS* simulator, high priority messages can be forwarded as soon as they arrive because the departure time can be immediately determined. However, a low priority message cannot be forward until simulated time in the logical process has advanced to the *departure* time (the time the hypercube *begins* sending the message) because it must first be determined that no high priority message will receive service ahead of it.

7.3 A Simulator with Poor Lookahead

The third simulator, *PREEMPT*, is identical to the *PRIORITY* simulator except that high priority messages preempt service of low priority messages. When a low priority message is preempted, it is assumed that the message must be completely resent once no other high priority messages remain that are waiting to use the link. The simulator for this system cannot forward a message to another logical process until simulated time has advanced to the *arrival* time (the time the *tail* of the message reaches the receiving hypercube node), so it has even poorer lookahead properties than the preceding simulator.

7.4 Performance Results

The hypercube simulations were performed on the Butterfly, and compared with execution of the sequential event list implementation. Unlike the previous experiments, these were performed on the Butterfly *Plus*, an upgraded version of the Butterfly that features 32 bit data paths (the original Butterfly has 16 bit data paths). The switch remains the same, so this effectively increases the cost of interprocessor communications. Because the simulation testbed already minimizes interprocessor communication, no program modifications were required. Experiments indicated that this hardware modification did not significantly affect the speedup measures derived earlier.

Overhead for these three simulators is shown in figures 14 and 15 for hypercubes of dimensions 4 and 6 (16 and 64 nodes respectively). Eight processors were used in these experiments. Upon reaching its destination, each message is assigned a high priority with probability P_{hprio}. In these experiments, P_{hprio} was selected to be either 0.01 or 0.50.

As predicted, the observed overhead steadily increases as the lookahead properties of the simulation are diminished. This is reflected in higher null message ratios in the deadlock avoidance simulator, and a larger message population required to induce avalanche in the detection and recovery simulator. Overheads are generally lower in the dimension four hypercube than the cube of dimension six for a fixed message population (as measured in messages per process) because there are fewer communication links; the simulators operate at peak efficiency when there is at least one message on each incoming link because no blocking occurs.

The lookahead properties of the simulator increase as P_{hprio} increases because more high priority messages are generated that can be forwarded as soon as they are received. This explains the lower overheads that were observed when P_{hprio} was increased.

Speedup curves for the hypercube simulators are shown in figures 16 and 17. Using eight processors, the parallel simulator executed anywhere from 5.7 times faster to nearly 20 times *slower* than the splay tree simulator, depending on the lookahead properties of the application. Some data points for very high message populations are missing because insufficient memory was available on a single processor to conduct an event list simulation.

The hypercube simulations provide additional evidence to support our contention that lookahead properties of the application are crucial to obtaining efficient performance for simulators using the deadlock avoidance and deadlock detection and recovery strategies. While the queueing network simulations demonstrated that it is possible to obtain dramatic speedups by reprogramming the simulation to fully exploit its lookahead properties, these experiments demonstrate that some simulations *inherently* contain poor lookahead, and cannot be improved by reprogramming. Such simulations appear to be poorly suited for the conservative simulation algorithms using deadlock avoidance and deadlock detection and recovery techniques, except in a few special circumstances such as networks that contain no feedback loops.

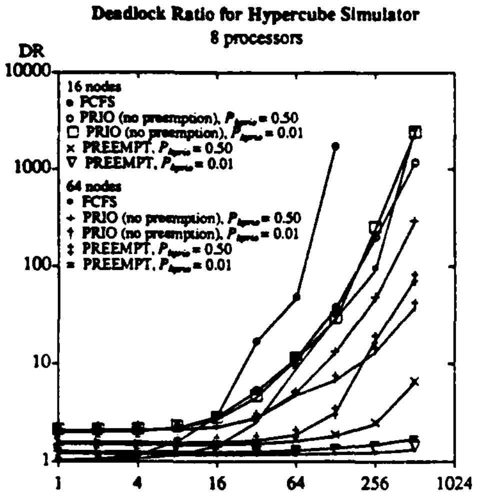

Figure 14. Overhead in hypercube simulator using deadlock recovery.

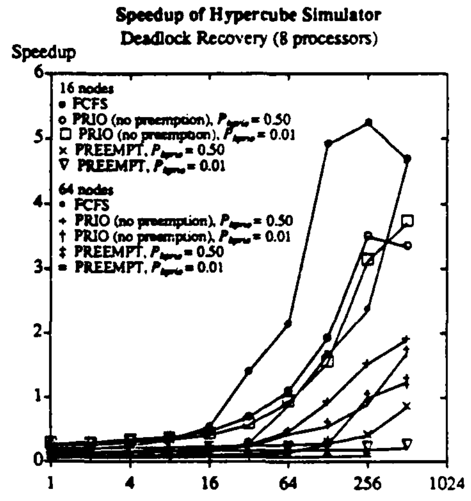

Figure 16. Speedup of hypercube simulator using deadlock recovery.

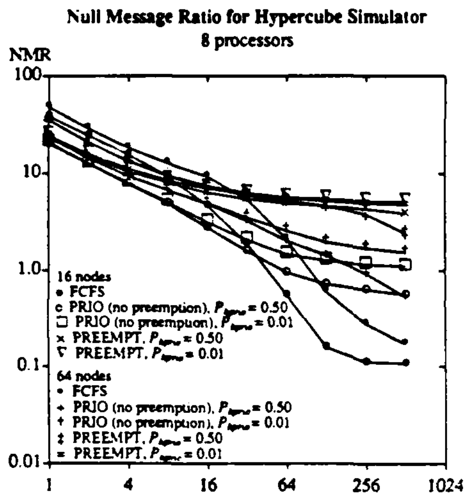

Figure 15. Overhead in hypercube simulator using deadlock avoidance.

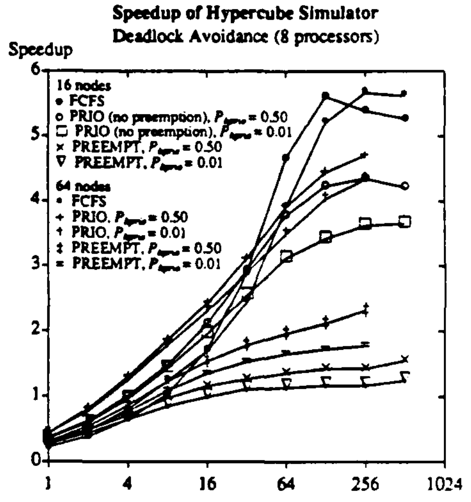

Figure 17. Speedup of hypercube simulator using deadlock avoidance

8. A Perspective on Lookahead: Non-Events

The influence of lookahead on performance can be viewed from another perspective: processes with very good lookahead ability are able to act in a largely autonomous fashion; their behavior is not heavily influenced by the activities of other processes, so they can perform simulation work at "full speed," limited only by the rate at which they can be fed work, and the number of CPU cycles (or other resources) that they can obtain. The optimized queueing network server process is a good example of such autonomous behavior.

On the other hand, processes with poor lookahead ability must frequently obtain additional information from other processes before they can safely proceed. This is unfortunate because not only must such processes wait for real events to be generated by other processes (corresponding to data dependencies that cannot be circumvented), but often they must also wait to be sure other events will *not* occur. The fact that an airplane will *not* crash and close the airport in the next moment of simulated time must be discovered before the airport process can go about its business of deciding what *will* happen next. We call these "phantom" events that never materialize *non-events*. Chandy and Misra recently captured these notions in an elegant formalism called *conditional* and *unconditional* knowledge [Chan87a].

In the deadlock avoidance simulator, knowledge of non-events is passed explicitly through the use of null messages. In the deadlock detection and recovery simulator, this information is obtained by system deadlock — processes with messages waiting to be processed must wait until they can be certain that specific events will *not* occur. Certainty as to the eventuality of non-events comes about when the deadlock is broken, and the deadlock resolution protocol is invoked. Sequential, event list simulators incur little or no overhead for non-events.

If non-events are possible, but occur infrequently, the simulator is often forced to wait needlessly, leading to very poor performance. The hypercube simulator containing preemption and *few* high priority messages is one example of such behavior. Optimistic simulation methods such as Time Warp appear to offer the greatest potential for addressing this problem, if the associated state saving and rollback overheads can be overcome.

9. Conclusions

Extensive empirical performance evaluations of distributed simulation programs were performed using the deadlock avoidance and deadlock detection and recovery algorithms developed by Chandy and Misra. The principal results of these studies are:

- The lookahead ability of logical processes plays a critical role in determining the efficiency of the deadlock avoidance and deadlock detection and recovery algorithms. This is attributed to the fact that processes must spend an excessive amount of time waiting to be sure that certain events will *not* occur if their lookahead ability is poor.
- Message avalanche was observed in the deadlock detection and recovery simulator for moderate to high message populations, and was necessary to achieve efficient execution. The poorer the lookahead ability of a process, the larger the message population necessary to achieve avalanche. If lookahead is sufficiently poor, avalanche may never be observed for workloads of practical interest.
- Deadlock detection and recovery simulators containing different types of logical processes can be adversely affected by a small number of processes that exhibit poor lookahead ability. The existence of a few such processes can greatly increase the message population necessary to achieve avalanche, even if many other processes contain very good lookahead properties. The deadlock avoidance simulator is not as severely affected by this behavior if the bulk of the simulation activity avoids processes with poor lookahead.
- Queueing networks that contain cycles, previously thought to be ill-suited for conservative distributed simulation algorithms, can achieve good performance if servers are reprogrammed to take advantage of all available lookahead.
- Simulation applications such as those containing infrequent preemptive events *inherently* have poor lookahead properties, and appear ill-suited for these algorithms. Applications containing state dependent behavior (e.g., load balancing mechanisms) similarly contain moderate to poor lookahead properties.
- Simulations of several hypercube-based communication networks with varying degrees of lookahead provide empirical data to support the above conclusions.

These studies demonstrate that parallel simulation algorithms can achieve significant speedups over sequential event list implementations if a moderate to high degree of parallelism is present, even if there are many feedback loops in the logical process topology. However, good lookahead properties are essential to obtaining good performance in simulations using deadlock avoidance or deadlock detection techniques. The fact that a few processes with poor lookahead properties can significantly degrade performance also limits the usefulness of these approaches.

Because conservative simulation algorithms must continually predict what will *not* happen in order to be able to safely proceed, these studies raise considerable doubt as to whether *any* conservative parallel simulation algorithm can obtain significant speedup in applications containing poor lookahead properties. In these situations, optimistic simulation algorithms such as Time Warp appear to offer much greater potential for achieving significant speedups.

REFERENCES

[Bile85a] W. Biles, "Statistical Considerations in Simulation on a Network of Microcomputers," *1985 Winter Simulation Conference Proceedings*, pp. 388-393 (December 1985).

[Chan83a] A. Chandak and J. C. Browne, "Vectorization of Discrete Event Simulation," *Proceedings of the 1983 International Conference on Parallel Processing*, pp. 359-361 (August 1983).

[Chan79a] K. M. Chandy and J. Misra, "Distributed Simulation: A Case Study in Design and Verification of Distributed Programs," *IEEE Transactions on Software Engineering* SE-5(5) pp. 440-452 (Sept. 1979).

[Chan81a] K. M. Chandy and J. Misra, "Asynchronous Distributed Simulation via a Sequence of Parallel Computations," *Communications of the ACM* 24(4) pp. 198-206 (April 1981).

[Chan87a] K. M. Chandy and J. Misra, "Conditional Knowledge as a Basis for Distributed Simulation," Technical Report 5251:TR:87, Computer Science Department, California Institute of Technology (1987).

[Comf84a] J. C. Comfort, "The Simulation of a Master-Slave Event Set Processor," *Simulation* 42(3) pp. 117-124 (March, 1984).

[Fuji87a] R. M. Fujimoto, "Performance Measurement of Distributed Simulation Strategies," Technical Report UUCS-87-026a, Computer Science Department, University of Utah, Salt Lake City, UT (November 1987).

[Fuji88a] R. M. Fujimoto, "Performance Measurement of Distributed Simulation Strategies," *Proceedings of the 1988 SCS Multiconference — Distributed Simulation, San Diego, California*, (February 1988).

[Jeff85a] D. R. Jefferson, "Virtual Time," *ACM Transactions on Programming Languages and Systems* 7(3) pp. 404-425 (July 1985).

[Jone86a] D. W. Jones, "An Empirical Comparison of Priority-Queue and Event-Set Implementations," *Communications of the ACM* 29(4) pp. 300-311 (April 1986).

[Kaud87a] F. J. Kaudel, "A Literature Survey on Distributed Discrete Event Simulation," *Simuletter* 18(2) pp. 11-21 (June 1987).

[Misr86a] J. Misra, "Distributed-Discrete Event Simulation," *ACM Computing Surveys* 18(1) pp. 39-65 (March 1986).

[Reed88a] D. A. Reed, A. D. Malony, and B. D. McCredie, "Parallel Discrete Event Simulation: A Shared Memory Approach," *IEEE Transactions on Software Engineering*, (to appear 1988).

[Seet79a] M. Seethalakshmi, "A Study and Analysis of Performance of Distributed Simulation," MS Report, University of Texas, Austin, Texas (May 1979).

[Slea85a] D. D. Sleator and R. E. Tarjan, "Self-Adjusting Binary Search Trees," *Journal of the ACM* 32(3) pp. 652-686 (July 1985).

[Sull77a] H. Sullivan and T. R. Bashkow, "A Large Scale, Homogeneous, Fully Distributed Parallel Machine," *Proceedings of the 4th Annual Symposium on Computer Architecture* 5(7) pp. 105-117 (March 1977).

[Thom86a] R. H. Thomas, "Behavior of the Butterfly Parallel Processor in the Presence pf Memory Hot Spots," *Proceedings of the 1986 International Conference on Parallel Processing*, pp. 46-50 (August 1986).